

# RIJKSINSTITUUT VOOR VISSERIJONDERZOEK

Haringkade 1 - Postbus 68 - 1970 AB IJmuiden - Tel.: +31 2550 64646

**Afdeling:** Technisch Onderzoek

**Rapport:** TO 90-04

Soortonderscheiding van schol en tong  
op grond van gegevens, verkregen met  
het FishVol - systeem.

**Auteur(s):** Guus van Riel.

**Project:** 156  
**Projectleider:** Drs. F.Storbeck  
**Datum van verschijnen:** Maart 1991

## Inhoud:

1	Inleiding.....	2
2	Probleemstelling.....	2
3	Voorbeeldprobleem en oplossing m.b.v. neuraal-netwerk.....	2
4.	Afleiding van de delta-regel.....	4
5.	Programmabeschrijving .....	9
	5.1 Representatie van het netwerk.....	9
	5.2 Gebruik van het programma.....	10
	5.3. Verwerking van de input.....	10
6	Testresultaten.....	11
7	Verdere ontwikkeling.....	14
8	Conclusie.....	14
9	Referenties .....	14
	appendices.....	15
	appendix 1 .....	15
	appendix 2 .....	19
	appendix 3 .....	21

**DIT RAPPORT MAG NIET GECITEERD WORDEN ZONDER TOESTEMMING VAN DE  
DIRECTEUR VAN HET R.I.V.O.**

## 1 INLEIDING

Als onderdeel van het EEG-project "*Integrated quality assurance of chilled food fish at sea*", nummer UP 1.67, heeft het RIVO onderzoek gedaan naar een alternatieve methode om het gewicht van (plat)vis te bepalen aan boord van kotterschepen. Dit onderzoeksproject wordt uitgevoerd door :

Technological Laboratory Ministry of Fisheries (Lingby	Denemarken)
Torry Research	(Aberdeen Schotland)
Instituut voor visproducten TNO	(IJmuiden)
Rijksinstituut voor Visserijonderzoek	(IJmuiden)

Bepaling van het volume van platvis is reeds mogelijk. Zie hiervoor het FishVol-rapport TO 89-11 (Daan, Storbeck, 1989).

In aansluiting op de gewichtsbepaling is onderzocht of het mogelijk is met de daarbij verkregen gegevens over de vis ook de soort te bepalen ten einde te komen tot een volledig geautomatiseerd sorteersysteem.

## 2 PROBLEEMSTELLING

Het verwerken van vis aan boord van schepen zoals dat nu nog gebeurt brengt een grote werkbelasting met zich mee voor de bemanning. Er wordt gemiddeld twee uur gevisd waarna de vangst in ongeveer een half uur verwerkt wordt. Intussen zijn de netten al overboord voor de volgende trek. Men werkt dus in een ritme van  $1\frac{1}{2}$  uur rust,  $\frac{1}{2}$  uur verwerking. Dit gebeurt meestal 4 dagen achtereen en 24 uur per dag. Een dergelijke dagindeling zal zeker vaak tot oververmoeidheid leiden, waardoor de kans op ongevallen op zee aanzienlijk toeneemt. Als de hele verwerking (dus ook het strippen) eenmaal volautomatisch kan gebeuren zal oververmoeidheid minder vaak voorkomen en de arbeidsomstandigheden op de vissersschepen hierdoor verbeteren omdat de bemanning op die manier aan de nodige rust toe kan komen. Iets wat mogelijk een verplichting wordt als in de toekomst de ARBO-wet ook voor zeegaande visserij schepen van toepassing is. Verder betekent automatisch sorteren op het schip (naar soort en gewicht) dat de vis aan de wal niet, zoals nu nog het geval is, andermaal van het ijs af moet voor deze handelingen.

De eerste stap moest zijn schol van tong onderscheiden. Om op grond van maatgegevens, verkregen met behulp van het bovengenoemde FishVol-systeem, onderscheid tussen deze vissoorten te kunnen maken is gekozen voor een zelflerend systeem op basis van een kunstmatig neurale netwerk. Een dergelijk systeem moet na een leerperiode in staat zijn in **real-time** vis te herkennen. Dit houdt in dat vanaf het moment dat de gegevens van de vis opgenomen zijn, het systeem binnen een fractie van een seconde bepaalt om welke soort het gaat en tot welke gewichtsklasse het exemplaar behoort.

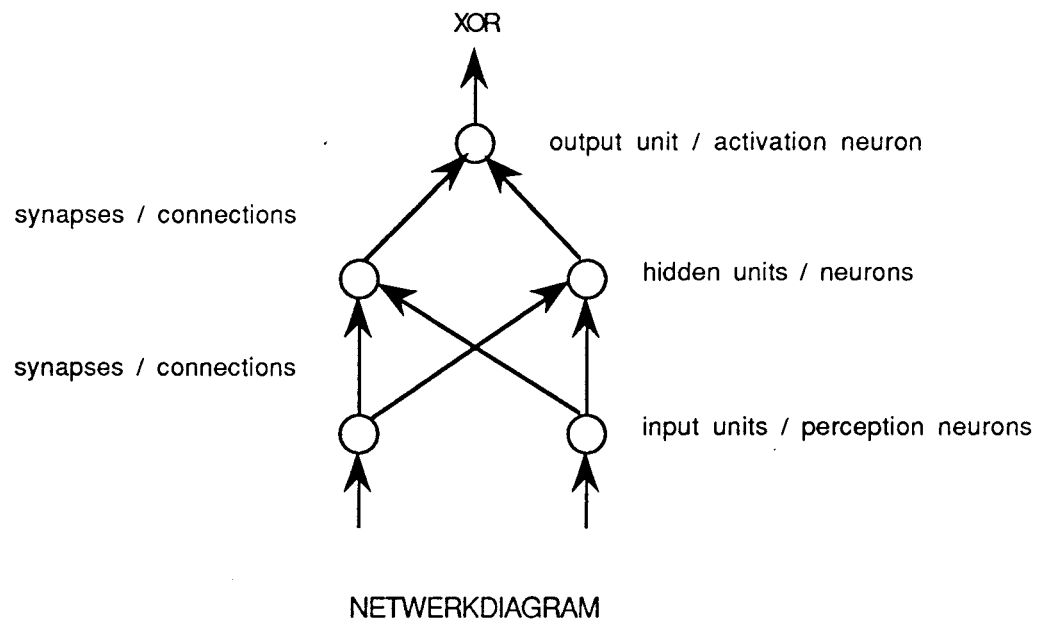
## 3 VOORBEELDPROBLEEM EN OPLOSSING M.B.V. NEURALE NETWERK

Het probleem is de bepaling van de logische "exclusieve of" (XOR)-waarde van een aantal waarden-paren, dat wil zeggen dat na aanbieding van een der vectoren  $\{(0,0), (0,1), (1,0), (1,1)\}$  door het systeem onmiddellijk de xor-waarde daarvan gegeven moet worden. De input en bijbehorende gewenste output staat weergegeven in de xor-tabel op de volgende pagina.

"XOR -waarheidstabel" :

INPUT- VECTOR	GEWENSTE OUTPUT
( 0 , 0 )	0
( 0 , 1 )	1
( 1 , 0 )	1
( 1 , 1 )	0

Dit probleem kan opgelost worden met behulp van een neurale netwerk. Een hiervoor geschikt neurale netwerk wordt grafisch voorgesteld als in onderstaande figuur (3.1) :



**figuur 3.1**

Elke verbinding (synapse in het diagram) krijgt initieel een "gewicht" toegekend. Elke input unit (onderste rij cirkels in het diagram) krijgt één van de elementen uit één der vectoren van de set  $\{(0,0) , (0,1) , (1,0) , (1,1)\}$  als input waarde. Voor input units geldt: output = input. De output waarde van de "hoger" gelegen (hidden en output) units is een functie van de outputwaarden van de een nivo "lager" liggende units en een daaraan gekoppelde gewichtsfactor.

In formulevorm weer te geven als : 
$$o_j = \sum_i w_{ji} o_i$$
 (Waarbij aangetekend dient te worden dat voor gebruik in de praktijk deze functie een nadere definitie behoeft.)

Tijdens het "leerproces" wordt de set input vectoren (later leerset genoemd) een aantal malen aan het net aangeboden. Wanneer de bij een aangeboden input **p** geproduceerde (actual) output niet gelijk is aan de bij **p** behorende gewenste (desired) output, dan "reageert" het systeem daarop met een verandering van de synapsegewichten. Dit proces wordt voortgezet tot geen gewichtsveranderingen meer nodig zijn om de correcte output van de gehele leerset te verkrijgen.

Appendix 1 bevat de broncode van een (van Pascal naar C vertaald) programma waarmee de werking van een neurale netwerk gedemonstreerd kan worden door het oplossen van het XOR-probleem.

Voor aanpassing van de gewichten wordt gebruik gemaakt van de **delta-regel** en het zgn. **back-propagation** algoritme. (Parker, 1989).

#### 4. AFLEIDING VAN DE DELTA-REGEL

Bij netwerken zonder hidden units wordt de standaard-vorm toegepast :

$$\Delta_p w_{ji} = \eta (t_{pj} - o_{pj}) i_{pi} = \eta \delta_{pj} i_{pi}$$

waarin :

- $\Delta_p w_{ji}$  : gewichtsverandering van de synapse tussen unit **i** en unit **j** bij input **p**.
- $\eta$  : factor ("learning rate") die ervoor zorgt dat de gewichtsverandering van een synapse per iteratie niet te groot wordt.
- $t_{pj}$  : de **gewenste** waarde van het **j-de** element van het outputpatroon  
bij inputpatroon **p**.
- $o_{pj}$  : de **actuele** waarde van het **j-de** element van het outputpatroon  
bij inputpatroon **p**.
- $i_{pi}$  : de waarde van het **i-de** element van het inputpatroon **p**.
- $\delta_{pj}$  : het **fout-signaal** van unit **u<sub>j</sub>** bij inputpatroon **p**.

Deze regel minimaliseert de kwadraatverschillen tussen de actuele en gewenste output-waarden, gesommeerd over alle output-units en alle paren input / output-vectoren.

$$\text{Zij} \quad E_p = \frac{1}{2} \sum_j (t_{pj} - o_{pj})^2 \quad (2)$$

de fout die optreedt bij input **p** en

$$\text{Zij} \quad E = \sum_p E_p$$

de totale fout over de hele verzameling inp./outp.-vectoren. (Later wordt deze verzameling de **leerset** genoemd).

De delta-regel levert een minimalisering van **E** op.

Als we schrijven : 
$$\frac{\partial E_p}{\partial w_{ji}} = \frac{\partial E_p}{\partial o_{pj}} \frac{\partial o_{pj}}{\partial w_{ji}} \quad (3)$$

dan geldt voor de eerste factor in het rechterlid :

$$\frac{\partial E_p}{\partial o_{pj}} = -(t_{pj} - o_{pj}) = -\delta_{pj} \quad (4)$$

De bijdrage van unit **j** aan  $E_p$  is evenredig met  $\delta_{pj}$  en omdat we eerst het werken met lineaire units beschouwen, geldt nu :

$$o_{pj} = \sum_i w_{ji} i_{pi} \quad (5)$$

zodat voor de tweede factor in het rechterlid van (3) geldt :

$$\frac{\partial o_{pj}}{\partial w_{ji}} = i_{pi} \quad (6)$$

Door substitutie van (4) en (6) in (3) krijgen we :

$$-\frac{\partial E_p}{\partial w_{ji}} = \delta_{pj} * i_{pi}$$

In combinatie met :

$$\frac{\partial E}{\partial w_{ji}} = \sum_p \frac{\partial E_p}{\partial w_{ji}}$$

leidt dat tot de conclusie dat de verandering van synapse-gewichten na een cyclus door de hele leersset evenredig is met deze afgeleide en dus dat de delta-regel  $E$  minimaliseert. Door gewichtsveranderingen na ieder aangeboden inp/oupt.-vectorpaar bestaat de kans verder van het gewenste minimum verwijderd te raken. Door de learning rate,  $\eta$ , klein genoeg te kiezen is dat gevaar echter voor zeer veel toepassingen verwaarloosbaar klein en levert de delta-regel een goede benadering van de minimalisering van  $E$  (Rumelhart, pg 324).

Voor gebruik in netwerken met hidden units (later semi-lineaire units genoemd) heeft de delta-regel de volgende "gegeneraliseerde" vorm :

$$\Delta_p w_{ji} = \eta (t_{pj} - o_{pj}) o_{pi} = \eta \delta_{pj} o_{pi}$$

waarin :

- $\Delta_p w_{ji}$  : de gewichtsverandering van de synapse tussen unit **i** en unit **j** bij input **p**.
- $\eta$  : de factor (de "learning rate") die ervoor zorgt dat de gewichtsverandering van een synapse per iteratie niet te groot wordt.
- $t_{pj}$  : de **gewenste** waarde van het j-de element van het outputpatroon bij input **p**.
- $o_{pj}$  : de **actuele** waarde van het j-de element van het outputpatroon bij input **p**.
- $o_{pi}$  : de waarde van het i-de element van het outputpatroon bij input **p**.

$\delta_{pj}$  : het fout-sigitaal van unit  $u_j$  bij inputpatroon  $p$ .

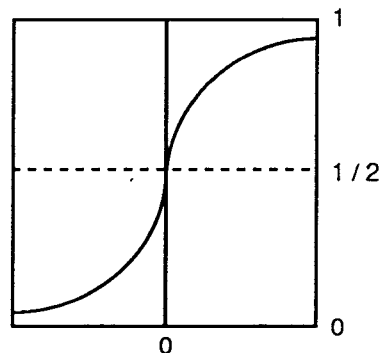
NB: voor een input-unit is  $o_{pi} = i_{pi}$

Bij het XOR-voorbeeld uit **fig 3.1** maken we gebruik van een netwerk met één verborgen laag. De input-vector wordt toegevoerd aan de onderste laag waarna de **actuele** output van hogere lagen berekend wordt aan de hand van reeds berekende (voor input-units bekende) output van de voorgaande laag.

Voor berekening van die output-waarden wordt gebruik gemaakt van de (semi-lineaire) activeringsfunctie :

$$o_{pj} = \frac{1}{1 + e^{-\sum_i w_{ji} o_{pi}}} \quad (\text{"logistieke" functie})$$

die grafisch kan worden weergegeven met :



**fig. 3.2** logistieke functie

waarin de output van unit  $j$  (noem  $o_{pj}$  op verticale as) een (monotoon stijgende, continu differentiëerbare) functie is van de totale output van het net (noem  $net_{pj} = \sum_i w_{ji} o_{pi}$  op horizontale as) náár unit  $j$ .

Het doel van de hele operatie is zodanige waarden voor alle  $w_{ji}$  te vinden dat de afgeleide van de logistieke functie zo klein mogelijk is met als gevolg dat de output-vector-waarden allemaal dicht bij 0 of dicht bij 1 zullen liggen (zoals aan de grafiek te zien is) en dus de van  $w_{ji}$  afhankelijke fout  $E$  zo klein mogelijk wordt.

Voor de logistieke functie wordt gedefinieerd :

$$net_{pj} = \sum_i w_{ji} o_{pi} \quad (7)$$

waarin wederom  $o(utput)_{pi} = i(nput)_{pi}$  als unit  $i$  een input-unit is.

Dus : 
$$o_{pj} = f_j(\text{net}_{pj}) \quad (8)$$

waarbij  $f$  niet-dalend en differentiëerbaar is.

Voor de juiste generalisatie van de delta-regel moeten we zorgen dat de verandering van synapse-gewicht tussen unit  $j$  en  $i$  bij input / output-patroon  $p$  (noem :  $\Delta_p w_{ji}$ ) evenredig is met de partiële afgeleide van de fout bij input / output-patroon  $p$  (noem :  $E_p$ ) naar  $w_{ji}$  ofwel :

$$\Delta_p w_{ji} \propto - \frac{\partial E_p}{\partial w_{ji}}$$

waarin  $E_p$  dezelfde fout-functie is als bij lineaire units. Evenals bij lineaire units kunnen we

schrijven : 
$$\frac{\partial E_p}{\partial w_{ji}} = \frac{\partial E_p}{\partial \text{net}_{pj}} \frac{\partial \text{net}_{pj}}{\partial w_{ji}} \quad (9)$$

Uit (7) volgt : 
$$\frac{\partial \text{net}_{pj}}{\partial w_{ji}} = \frac{\partial}{\partial w_{ji}} \sum_k w_{jk} o_{pk} = o_{pi}$$

Als we definiëren : 
$$\delta_{pj} = - \frac{\partial E_p}{\partial \text{net}_{pj}}$$

komt dat overeen met de definitie van  $\delta_{pj}$  in de standaard delta-regel voor lineaire units, immers :  $\text{net}_{pj} = o_{pj}$  als  $u_j$  lineair is. (vergelijking (4)). Dus (9) kan geschreven worden als :

$$- \frac{\partial E_p}{\partial w_{ji}} = \delta_{pj} o_{pi}$$

Dit laatste betekent dat we de gewichten moeten aanpassen volgens :

$$\Delta_p w_{ji} = \eta \delta_{pj} o_{pi} \quad (11)$$

Hetzelfde dus als bij de standaard delta-regel. Het probleem is hier echter : hoe wordt  $\delta_{pj}$  bepaald voor elke unit in het netwerk ?

Om  $\delta_{pj} (= - \frac{\partial E_p}{\partial \text{net}_{pj}})$  te bepalen wordt, gebruik makend van de kettingregel, deze

uitdrukking geschreven als produkt van twee factoren :

$$\delta_{pj} = - \frac{\partial E_p}{\partial \text{net}_{pj}} = - \frac{\partial E_p}{\partial o_{pj}} \frac{\partial o_{pj}}{\partial \text{net}_{pj}} \quad (12)$$

Uit (8) volgt :  $\frac{\partial o_{pj}}{\partial \text{net}_{pj}} = f'_j(\text{net}_{pj})$

ofwel de afgeleide van de (semi-lineaire) functie **f** waarmee de output van een unit bepaald wordt.

Als  $u_j$  een **output**-unit is dan (volgens definitie van  $E_p$ ) :

$$\frac{\partial E_p}{\partial o_{pj}} = -(t_{pj} - o_{pj})$$

Substitutie in (12) geeft :

$$\delta_{pj} = -(t_{pj} - o_{pj}) f'_j(\text{net}_{pj}) \quad (13)$$

Als  $u_j$  een **hidden**-unit is, maken we weer gebruik van de kettingregel om te schrijven:

$$\begin{aligned} \frac{\partial E_p}{\partial o_{pj}} &= \sum_k \frac{\partial E_p}{\partial \text{net}_{pk}} \frac{\partial \text{net}_{pk}}{\partial o_{pj}} = \sum_k \frac{\partial E_p}{\partial \text{net}_{pk}} \frac{\partial}{\partial o_{pj}} \sum_i w_{ki} o_{pi} = \\ &= \sum_k \frac{\partial E_p}{\partial \text{net}_{pk}} w_{kj} = - \sum_k \delta_{pk} w_{kj} \end{aligned}$$

Substitutie in (12) geeft dan :

$$\delta_{pj} = f'_j(\text{net}_{pj}) \sum_k \delta_{pk} w_{kj} \quad (14)$$

(13) en (14) bieden een recursieve procedure om de  $\delta$  van elke unit in het net te berekenen. Deze  $\delta$ 's kunnen dan in (11) gebruikt worden om de synapse-gewichten aan te passen.

Dit geheel is samen te vatten in drie vergelijkingen :

1)  $\Delta_p w_{ji} = \eta \delta_{pj} o_{pi}$  : de verandering van een synapse-gewicht is evenredig met het produkt van een fout-sigitaal,  $\delta$ , behorend bij de doel-unit van die synapse en de output,  $o_{pi}$ , behorend bij de bron-unit.

2)  $\delta_{pj} = -(t_{pj} - o_{pj}) f'_j(\text{net}_{pj})$  : het fout-sigitaal van een **output**-unit.

3)  $\delta_{pj} = f'_j(\text{net}_{pj}) \sum_k \delta_{pk} w_{kj}$  : het fout-sigitaal van een **hidden**-unit.

Toepassing van de delta-regel vereist dus twee bewerkingsfasen :



Eerst wordt de input(vector) aangeboden en voorwaarts verspreid ("forward propagated") over het netwerk om een (actuele) output-waarde  $o_{pj}$  te berekenen voor elke unit en in het bijzonder voor de output-units. Vergelijking hiervan met de gewenste output ("target-vector") levert een  $\delta_{pj}$  op voor elke output-unit.

In de tweede fase wordt het fout-sigitaal achterwaarts verspreid ("back propagated") over het netwerk en worden de nodige gewichtsaanpassingen gerealiseerd middels de delta-regel. Tijdens deze achterwaartse doorgang vindt steeds herberekening van  $\delta$  plaats : eerst voor de output-units; vervolgens worden alle gewichten van de synapsen naar de outputlaag aangepast, dan herberekening van  $\delta$ 's in de voorlaatste laag enzovoort.

Een eigenschap van de logistieke functie is dat de extreme waarden (0 en 1) alleen bereikt worden bij (in absolute waarde) oneindig grote gewichten van de synapsen (te zien aan de grafiek). Daarom zal in een situatie waarbij de gewenste output-waarde een element uit  $\{0,1\}$  is, het systeem nooit die waarden echt bereiken.

Daarom wordt in zo'n geval een maximaal te accepteren waarde van  $E$  (het **convergentie-criterium**) aan het systeem meegegeven.

De snelheid waarmee het systeem leert is (o.a.) afhankelijk van de eerder genoemde "learning rate" ( $\eta$  in de deltaregel). Hoe groter  $\eta$ , hoe sneller het leerproces zal verlopen maar wordt  $\eta$  te groot gekozen dan zullen de  $o_{pj}$  gaan "verspringen" van dicht bij 0 naar dicht bij 1, maar nooit dicht genoeg bij 0 of 1 om het netwerk aan het convergentie-criterium te laten voldoen.

In de praktijk blijkt  $0.25 \leq \eta \leq 0.40$  goed te voldoen (Rumelhart 1986).

## 5. PROGRAMMABESCHRIJVING

### 5.1 Representatie van het netwerk

Om het neurale netwerk te representeren wordt gebruik gemaakt van twee gelinkte (voor de Lessons en Synapsen) en een dubbel gelinkte lijst (voor de Neurons) van structures.

De lessen worden vanuit een (lesson.in)-file in een gelinkte lijst ingelezen die als geheel de zgn. **leerset** vormt.

Een LESSON - structure bevat de volgende velden :

- id nummer.
- error (verschil tussen actuele en gewenste output)
- perception (lijst van inputwaarden)
- activation (lijst van gewenste outputwaarden van het net)
- pointer naar volgende les in de set.
- twee besturingsbooleans.

Een NEURON - structure bevat de velden :

- id nummer.
- type (perceptron, hidden neuron of activator)
- value.
- error.

Een SYNAPSE - structure bestaat uit :

- pointer naar een gelinkte lijst van synapsen.
- pointers naar volgend en vorig neuron in dubbel gelinkte lijst.
- id nummer.
- pointer naar neuron waar de synapse naartoe leidt.
- gewicht.
- pointer naar volgende synapse in de gelinkte lijst.

## 5.2 Gebruik van het programma

Het programma biedt mogelijkheden om naar wens en behoefte een netwerk te laten genereren en leersets aan te bieden en te bewaren.

Opgegeven kan worden:

- \* Het aantal input en output units, evenals het aantal hidden units.
- \* De file waarin de te gebruiken leerset is opgenomen.
- \* De file waarin een te gebruiken netwerk is opgenomen.
- \* Het convergentie-criterium.
- \* Het maximum aantal iteraties dat gemaakt mag worden per leeracyclus.
- \* De learning rate  $\eta$ .

Verder kan men:

- \* Te lichte synapsen verwijderen waarna ook eventueel overbodig geworden neuronen (zonder verbinding met enig ander neuron) worden "weggesneden"
- \* Ook het minimumgewicht van de synapsen kan worden ingesteld. Dit alles om te testen of een vereenvoudigd netwerk ook voldoet aan de te stellen eisen. "Overvolle" netwerken blijken zich in de praktijk soms vreemd te gedragen.
- \* Een reeds doorgerekend netwerk bewaren zodat op een later tijdstip het systeem "verder kan leren"
- \* Kiezen voor wel of geen tussenresultaten en hoe vaak men die wil zien.

## 5.3. Verwerking van de input

( De broncode van de in deze paragraaf genoemde functions is opgenomen in appendix 2 )

Voor verwerking van de input kan een netwerk gegenereerd of ingelezen worden met de benodigde neuronen en (indien gegenereerd random-) waarden voor de synapse-gewichten.

De lessen uit de set worden vervolgens in willekeurige volgorde aan het netwerk aangeboden, waarna de volgende bewerkingen erop worden uitgevoerd:

- A) De perceptionwaarden uit een les-structure worden geplaatst in het value-veld van een perceptron. Dit gebeurt in de function TriggerPerceptrons() in de module Learn.c.
- B) De function DoCalculate() in dezelfde module bepaalt de gewogen som van de outputwaarden per neuron m.b.v. de logistieke functie.

- C) `ObserveActivity()` in module `Learn.c` berekent de actuele output-vector van de onderhavige "les".
- D) Nu moeten in achterwaartse richting (beginnend bij de Activators tot aan de Perceptrons) alle errorwaarden en synapse-gewichten weer worden aangepast. (Hier vindt dus feitelijk het `BackPropagation`-proces plaats)  
Dit gebeurt in function `DoBackPropagation()` (eveneens in module `Learn.c`) door :

van elk neuron op huidig nivo

- error te vervangen door het produkt :  

$$\text{error} * \text{output van het neuron} * (1 - \text{output van het neuron}).$$
 ( Dit is het produkt van error huidig neuron en de eerder genoemde partiële afgeleide van de logistieke functie:  $\delta_{pj} * o_{pj} * (1 - o_{pj})$  )

van alle "aanhangende" synapsen

- gewicht te vervangen door het produkt :  

$$\text{learnrate} * \text{error} * \text{output van het "aanhangende" neuron}.$$
 ( Overeenkomend met:  $\eta * \delta_{pj} * o_{pi}$  )

en van alle "aanhangende" neuronen

- error te vervangen door de som  

$$\text{error} + (\text{gewicht van synapse} * \text{error van neuron op huidig nivo})$$
 ( Komt overeen met:  $\delta_{pi} + (w_{ij} * \delta_{pj})$  )

Als na deze acties het kwadraat van de in de les resterende totale fout (`errorSquared`) kleiner is dan de vereiste tolerantie dan wordt deze les als geconvergeerd beschouwd. Wanneer alle lessen uit de leer-set geconvergeerd zijn, dan is het leerproces succesvol geweest en heet het hele netwerk geconvergeerd te zijn binnen de criteria.

## 6 TESTRESULTATEN

De aangeboden leerset bestond uit 8 schollen en 10 tongen van verschillende maat. Van deze vissen werden de volgende maatgegevens bepaald:

De breedte op 10, 30, 50, 70 en 90 procent van de lengte (genormaliseerd naar de lengte van de betreffende vis) en de bijbehorende diktes.

Het gebruikte netwerk bestond derhalve uit **10** perceptrons, **10** hidden neurons en **1** activator (één activator voldeed omdat alleen bepaald moest worden of het schol of tong betrof). Het leerproces verliep in deze opzet bevredigend.

Bij het bekijken van de gegevens in de lesson file leken (op het oog) de waarden van slechts één breedte met bijbehorende dikte reeds voldoende significant om het gewenste onderscheid te kunnen maken. Er is toen gekozen voor de breedte en bijbehorende dikte op 45% vanaf de kop van de vis; daar bleek nl. gemiddeld de grootste breedte te liggen bij zowel schol als tong.

Het net (met deze keer 2 perceptrons, 2 hidden nodes en 1 activator), convergeerde wel met deze input, maar het leren duurde relatief lang en de resultaten van een test met 14 tongen en 15 schollen waren niet nauwkeurig genoeg: drie schollen en twee "niet-schollen" werden niet als zodanig herkend.

De volgende keuze was een net met 6 perceptrons, 6 hidden nodes en 1 activator waaraan een leerset van 8 schollen en 10 tongen (zie de tabel **Leerset** op de volgende pagina) werd aangeboden. Hierin zijn de perception-waarden tussen 0 en 1 gebracht (zie appendix 3). Dit leverde een zeer snel leerproces waarvoor minder dan 3000 iteraties nodig waren.

Daarna werd, gebruik makend van het geconvergeerde net, van 15 schollen en 14 tongen de soort bepaald (zie de tabel **Testset** op de volgende pagina) Hoe kleiner *error na calculatie* (in ieder geval  $< 0.5$  want dat is als norm genomen voor juiste vaststelling van de soort), hoe beter het resultaat. Uit de voorlaatste kolom blijkt dan dat één tong (test 22) niet als tong herkend is. Met een nauwkeurigheid van 96.5% haalt deze test de vooraf gestelde eis van 95%.

Leersset van 8 schollen (Action = 1.0) en 10 tongen (Action = 0.0) :

Les	perceptron						Action	error na convergentie
	1	2	3	4	5	6		
1	0.64	0.28	0.19	0.35	0.07	0.19	1.0	0.0027
2	0.73	0.37	0.49	0.52	0.15	0.24	1.0	0.0031
3	0.61	0.41	0.34	0.55	0.12	0.29	1.0	0.0009
4	0.65	0.26	0.19	0.20	0.07	0.10	1.0	0.0154
5	0.60	0.35	0.38	0.46	0.15	0.28	1.0	0.0059
6	0.75	0.44	0.60	0.57	0.35	0.30	1.0	0.0491
7	0.74	0.38	0.88	0.68	0.22	0.29	1.0	0.0268
8	0.78	0.38	0.53	0.62	0.13	0.36	1.0	0.0007
9	0.40	0.17	0.44	0.32	0.24	0.18	0.0	0.0067
10	0.44	0.26	0.44	0.36	0.30	0.15	0.0	0.0103
11	0.38	0.26	0.41	0.36	0.27	0.19	0.0	0.0193
12	0.46	0.19	0.37	0.27	0.22	0.17	0.0	0.0389
13	0.57	0.20	0.52	0.23	0.24	0.18	0.0	0.0053
14	0.55	0.27	0.52	0.42	0.31	0.18	0.0	0.0506
15	0.49	0.22	0.57	0.31	0.39	0.16	0.0	0.0000
16	0.51	0.18	0.59	0.31	0.36	0.16	0.0	0.0001
17	0.57	0.19	0.53	0.18	0.32	0.12	0.0	0.0001
18	0.46	0.16	0.45	0.05	0.15	0.07	0.0	0.0005

Testset van 15 schollen (Action = 1.0) en 14 tongen (Action = 0.0).

Test	perceptron						Action	error na calculatie	+/-
	1	2	3	4	5	6			
1	0.77	0.33	0.46	0.60	0.17	0.34	1.0	0.0013	++
2	0.50	0.37	1.06	0.72	0.21	0.38	1.0	0.4462	+
3	0.36	0.38	0.36	0.66	0.10	0.29	1.0	0.0021	++
4	0.67	0.44	0.50	0.80	0.19	0.36	1.0	0.0005	++
5	0.58	0.38	0.14	0.60	0.11	0.36	1.0	0.0004	++
6	0.36	0.40	0.34	0.59	0.16	0.28	1.0	0.0064	++
7	0.52	0.42	0.25	0.68	0.16	0.31	1.0	0.0007	++
8	0.58	0.45	0.67	0.86	0.36	0.40	1.0	0.0111	++
9	0.65	0.43	0.54	0.70	0.25	0.37	1.0	0.0024	++
10	0.56	0.44	0.34	0.59	0.13	0.32	1.0	0.0009	++
11	0.49	0.39	0.32	0.70	0.15	0.37	1.0	0.0008	++
12	0.57	0.39	0.34	0.70	0.12	0.32	1.0	0.0005	++
13	0.60	0.34	0.38	0.56	0.20	0.32	1.0	0.0048	++
14	0.48	0.44	0.45	0.70	0.24	0.36	1.0	0.0039	++
15	0.43	0.37	0.28	0.72	0.14	0.34	1.0	0.0009	++
16	0.42	0.33	0.52	0.62	0.38	0.29	0.0	0.1579	+
17	0.41	0.29	0.51	0.54	0.36	0.32	0.0	0.0599	++
18	0.36	0.20	0.36	0.27	0.26	0.20	0.0	0.0052	++
19	0.47	0.28	0.63	0.72	0.45	0.34	0.0	0.0415	++
20	0.31	0.24	0.34	0.38	0.26	0.22	0.0	0.0431	++
21	0.43	0.34	0.51	0.50	0.36	0.27	0.0	0.0660	++
22	0.51	0.30	0.46	0.63	0.31	0.30	0.0	0.8732	---
23	0.56	0.43	0.59	0.60	0.42	0.29	0.0	0.3280	+
24	0.38	0.24	0.40	0.42	0.28	0.22	0.0	0.0510	++
25	0.42	0.27	0.39	0.43	0.25	0.24	0.0	0.3437	+
26	0.44	0.25	0.58	0.45	0.41	0.23	0.0	0.0004	++
27	0.48	0.25	0.44	0.36	0.31	0.23	0.0	0.0186	++
28	0.37	0.22	0.51	0.40	0.30	0.21	0.0	0.0033	++
29	0.29	0.21	0.33	0.36	0.20	0.23	0.0	0.1114	+

## **7 VERDERE ONTWIKKELING**

Met het tot nu toe ontwikkelde systeem is het mogelijk om schol te selecteren uit een mengsel van schol en tong op grond van contourgegevens. De vis moet echter nog goed recht, de rug naar boven en met de kop in de looprichting van de band, onder de laserstraal doorgevoerd worden om bruikbare gegevens op te leveren omdat ook de leerset op deze wijze tot stand is gekomen. Dit zou opgelost kunnen worden door ontwikkeling van apparatuur die de vis in de juiste positie op de transportband kan leggen.

Het scan-systeem aanpassen voor in willekeurige positie liggende vissen zou kunnen door meer breedtelijnen uit het beeld op te nemen waaruit dan een hoofdas berekend wordt aan de hand waarvan -voor de soort significante- contourgegevens gegenereerd worden.

## **8 CONCLUSIE**

Voor herkenning van schol en tong lijkt het hier beschreven systeem bruikbaar. Omdat het netwerk steeds kan "bijleren" en dus waarschijnlijk verfijnder kan gaan beoordelen mag verwacht worden dat ook andere platvissorten zoals bijvoorbeeld schol en schar (die qua contouren erg op elkaar lijken) onderscheiden kunnen gaan worden.

## **9 REFERENTIES**

- Daan, B., F. Storbeck.(1989), TO 89-11 Het FishVol-systeem: Bepaling van visgewicht met behulp van beeldverwerking in samenhang met gestructureerd licht, Rijksinstituut voor viserijonderzoek.
- Parker, Dave (1989) , Parker's Perceptions. In: Dr. Dobb's Journal, october 1989.vol 14, pp.112-147
- Rumelhart, D.E., G.E.Hinton and R.J.Williams (1986b), Learning Internal Representations by Error Propagation. In: Rumelhart, D. and J.L. McClelland (eds.), Parallel Distributed Processing, vol 1, pp.318-364, MIT Press

## APPENDICES

### appendix 1

```

/*****
 *      Netwerksimulatieprogramma ter bepaling van AND en XOR
 *****/
#include "defs.h"
#define      MAXITERATIONS  20000

CellRecord  CellArray[NumOfRows+1][NumOfCols+1];
double      inputs[NumOfCols+1];
double      desiredOutputs[NumOfCols+1];

void CalculateInputsAndOutputs(iteration)
int iteration;
/* Bereken de inputs en de gewenste outputs voor de huidige iteratie */
/* De inputs bestaan afwisselend uit de 4 patronen (0.05,0.05), (0.95,0.05), */
/* (0.05,0.95), (0.95,0.95). De gewenste outputs (target-vectoren) zijn */
/* (0.05,0.05), (0.05,0.95), (0.05,0.95), (0.95,0.05). Het eerste element */
/* van de target-vector is de logische AND van de inputvector, het tweede */
/* element is de logische XOR van de inputvector. */
{
    if ((iteration % 2) == 1 ) {
        inputs[1] = One;
    }
    else {
        inputs[1] = Zero;
    }
    if ((iteration % 4) > 1) {
        inputs[2] = One;
    }
    else {
        inputs[2] = Zero;
    }
    if (And(inputs[1], inputs[2])) {
        desiredOutputs[1] = One;
    }
    else {
        desiredOutputs[1] = Zero;
    }
    if (Xor(inputs[1], inputs[2])) {
        desiredOutputs[2] = One;
    }
    else {
        desiredOutputs[2] = Zero;
    }
}
}
```

```

void UpdateCellOnForwardPass(row,column)
int    row, column;
/* Bereken de output van de cel op de aangegeven row en column */
{
    int    j;
    double Sum;
    Sum = 0.0;          /* initialiseer gewogen som van inputs */
    for (j=0; j <= NumOfCols; j++) {
        /* Bereken de gewogen som van inputs */
        Sum += CellArray[row][column].weights[j] *
            CellArray[row-1][j].output;
    }
    CellArray[row][column].output = 1.0/(1.0+exp((double)-Sum));
    /* Bereken de output van de cel met logistieke functie*/
    CellArray[row][column].error = 0.0;
    /* Wis de foutwaarde voor achterwaartse propagatie*/
}

void UpdateCellOnBackwardPass(row, column)
int    row, column;
/* Bereken foutsignalen en pas gewichten aan in de achterwaartse beweging */
{
    int    j;

    for (j = 1; j <= NumOfCols; j++) {
        /* Propageer de fout naar de cellen "onder" de huidige */
        CellArray[row-1][j].error =
            CellArray[row-1][j].error +
            CellArray[row][column].error *
            CellArray[row][column].output *
            (1.0-CellArray[row][column].output) *
            CellArray[row][column].weights[j];
    }
    for (j=0; j <= NumOfCols; j++) {
        /* Pas de gewichten in de huidige cel aan */
        CellArray[row][column].weights[j] =
            CellArray[row][column].weights[j] +
            LearningRate*CellArray[row][column].error *
            CellArray[row][column].output *
            (1.0-CellArray[row][column].output) * CellArray[row-1][j].output;
    }
}

main()
{
    int    i, j, k, t ;
            /* i telt rows, j telt columns en k telt gewichten */
    int    convergedIterations;
            /* Het netwerk moet gedurende 4 opvolgende iteraties */
            /* geconvergeerd blijven (1 voor elke input-vector). */
    int    iteration;
            /* Totaal tot nu toe uitgevoerde iteraties */
    double errorSquared;
            /* Kwadraat van de fout bij huidige iteratie*/
}

```



```

printf("\nIteration   Inputs   Desired Outputs   Actual Outputs\n\n");
iteration=0;          /* start bij iteratie 0 */
convergedIterations = 0;
                      /* Het netwerk is nog niet geconvergeerd */
for (i=1; i<=NumOfRows; i++) {
    for(j=1; j<=NumOfCols; j++) {
        for (k=0; k<=NumOfCols; k++) {
            CellArray[i][j].weights[k] = ((rand() % 1000) * 0.001);
        }
    }
}

for (i=0; i<=NumOfRows; i++) {
    /* Initialiseer de outputs van de ("dummy") constante cellen */
    CellArray[i][0].output = One;
}

do {
    CalculateInputsAndOutputs(iteration);
    for (j=1; j<=NumOfCols; j++) {
        /* Breng inputs in dummy inputcellen */
        CellArray[0][j].output = inputs[j];
    }

    for (i=1; i<=NumOfRows; i++) {
        /* Propageer input voorwaarts door netwerk */
        for (j=1; j<=NumOfCols; j++) {
            UpdateCellOnForwardPass(i,j);
        }
    }

    for (j=1; j<=NumOfCols; j++) {
        /* Bereken de fouten */
        CellArray[NumOfRows][j].error =
            desiredOutputs[j] - CellArray[NumOfRows][j].output;
    }

    for (i = NumOfRows; i>=1; i--) {
        /* Propageer fouten achterwaarts door het netwerk en pas de */
        /* gewichten aan */
        for (j = 1; j <= NumOfCols; j++) {
            UpdateCellOnBackwardPass(i,j);
        }
    }

    errorSquared = 0.0;
    /* wis errorSquared */
    for (j = 1; j <= NumOfCols; j++) {
        /* bereken nieuwe errorSquared */
        errorSquared = errorSquared +
            CellArray[NumOfRows][j].error *
            CellArray[NumOfRows][j].error;
    }
}

```

```

        if (errorSquared < Criteria) {
            /* als netwerk convergeerde, hoog convergedIterations op */
            convergedIterations++;
        }
        else {
            /* anders wordt convergedIterations weer 0 */
            convergedIterations = 0;
        }
        if (((iteration % 1000 < 4) || (iteration > 16600)) && (iteration <
                                                    MAXITERATIONS)) {
            printf("\n %5d  ",iteration);
            /* druk iteratienummer af */

            for (j = 1; j <= NumOfCols; j++) {
                /* druk inputvector af */
                printf("%4.2f  ",inputs[j]);
            }
            printf("    ");

            for (j = 1; j <= NumOfCols; j++) {
                /* druk targetvector af */
                printf("%4.2f  ",desiredOutputs[j]);
            }
            printf("    ");

            for (j = 1; j <= NumOfCols; j++) {
                /* druk actuele outputvector af */
                printf("%4.2f  ", CellArray[NumOfRows][j].output);
            }
            iteration++;
        } while ((convergedIterations != 4));
        /* Stop als netwerk convergeerde op alle 4 inputvectoren */
        /* of als het maximaal toegestane aantal iteraties is bereikt */
        printf("\nIteration   Inputs       Desired Outputs   Actual Outputs");

        if (convergedIterations != 4) {
            /* druk boodschap omtrent resultaat af */
            printf("\nNetwork did not converge");
            printf("\nNmbr of convergedIterations = %d", convergedIterations);
            printf("\nNmbr of iterations           = %d\n", iteration);
            printf("\nErrorSquared                = %8.5f", errorSquared);
        }
        else {
            printf("\nNetwork has converged to within criteria.");
            printf("\nNmbr of convergedIterations = %d", convergedIterations);
            printf("\nNumber of iterations = %d\n",iteration);
        }
    } /* end of main */

```

## appendix 2

```
/*
 * Copyright (c) 1990, Rijksinstituut voor Visserijonderzoek
 * All rights reserved
 * @(#)Learn.c      1.2 90/06/25
 */

#include "NNetDefs.h"
#define EXTERN
#include "NNetGlobals.h"

#define      dY(x) (x * (1.0 - x))
#define F(x) (1.0 / (1.0 + exp(-(double)(x))))

/**
 ** TriggerPerceptrons - Initialize values in perceptron neurons
 **/
void TriggerPerceptrons(LessonPtr theLessonPtr)
LessonPtr    theLessonPtr;
{
    register      int          i = 0;
    register      NeuronPtr    theNeuronPtr;

    for (i = 0, theNeuronPtr = theNeuronTailPtr;
        theNeuronPtr->type == Perceptron; theNeuronPtr = theNeuronPtr->prev) {
        theNeuronPtr->value = theLessonPtr->perception[i++];
    }
}

/**
 ** DoCalculate - calculates the neuron activations
 **/
void DoCalculate()
{
    register      NeuronPtr    theNeuronPtr;
    register      SynapsePtr    theSynapsePtr;
    float          sum;

    for (theNeuronPtr = theHiddenTailPtr; theNeuronPtr != NIL;
        theNeuronPtr = theNeuronPtr->prev) {
        sum = 0.0;
        for (theSynapsePtr = theNeuronPtr->synapseHeadPtr;
            theSynapsePtr != NIL; theSynapsePtr = theSynapsePtr->next) {
            sum +=
                (theSynapsePtr->neuronPtr)->value * theSynapsePtr->weight;
        }
        theNeuronPtr->value = F(sum);
    }
}
```

```

/**
** ObserveActivity - computes difference between activation values and
** requested values
**/
void ObserveActivity(theLessonPtr)
LessonPtr theLessonPtr;
{
    register    int        i = 0;
    register    NeuronPtr  theNeuronPtr;
    register    double     sumsq = 0.0;

    for (theNeuronPtr = theNeuronHeadPtr;
        theNeuronPtr->type == Activator; theNeuronPtr = theNeuronPtr->next) {
        theNeuronPtr->error = (theLessonPtr->activation[i++] -
            theNeuronPtr->value);
        sumsq += theNeuronPtr->error * theNeuronPtr->error;
    }
    theLessonPtr->error = sumsq;
}

/**
** DoBackPropagation - propagates the error backwards through the neurons
**/
void DoBackPropagation()
{
    register    NeuronPtr  theNeuronPtr;
    register    SynapsePtr theSynapsePtr;
    register    float      error, faultProp;

    for (theNeuronPtr = theNeuronHeadPtr;
        theNeuronPtr->type != Perceptron;
        theNeuronPtr = theNeuronPtr->next) {
        error = theNeuronPtr->error * dY(theNeuronPtr->value);
        faultProp = learnrate * error;
        for (theSynapsePtr = theNeuronPtr->synapseHeadPtr;
            theSynapsePtr != NIL; theSynapsePtr = theSynapsePtr->next) {
            theSynapsePtr->weight += faultProp *
                (theSynapsePtr->neuronPtr->value;
                (theSynapsePtr->neuronPtr->error += theSynapsePtr->weight *
                    error;
            }
        theNeuronPtr->error = 0.0;
    }
}

```

### appendix 3

Breedte en dikte van de vis zijn opgenomen met een frequentie van een frame per 0.44 cm lengte. Drie van deze breedtes met corresponderende diktes worden gebruikt. In het volgende programmafragment staat normFactor voor  $(1 / \text{lengte van de vis})$  waarbij lengte van de vis bepaald wordt door  $(\text{frameCnt} * 0.44 \text{ cm})$ .

```
LineNmb = (int)(frameCnt * 0.45);          /* lijn op 45% vanaf de kop */
fprintf(stderr, "LineNmb: %d\n", LineNmb );
fprintf(stderr, " %4.2f %4.2f\n", *(brAr + LineNmb), *(thAr + LineNmb));
fprintf(fdOut, "\n %13.6e %13.6e",
        (((*(brAr + LineNmb) * normFactor) - 0.2) * 3),
        (((*(thAr + LineNmb) * normFactor * 10.0) - 0.3) * 2));

/* In de laatste twee regels worden de te gebruiken inputwaarden tussen 0 en 1 gebracht. */

LineNmb = (int)(frameCnt * 0.7);          /* lijn op 70% vanaf de kop */
fprintf(stderr, "LineNmb: %d\n", LineNmb );
fprintf(stderr, " %4.2f %4.2f\n", *(brAr + LineNmb), *(thAr + LineNmb));
fprintf(fdOut, "\n %13.6e %13.6e",
        (((*(brAr + LineNmb) * normFactor) - 0.15) * 4),
        (((*(thAr + LineNmb) * normFactor * 10.0) - 0.2) * 5));

LineNmb = (int)(frameCnt * 0.8);          /* lijn op 80% vanaf de kop */
fprintf(stderr, "LineNmb: %d\n", LineNmb );
fprintf(stderr, " %4.2f %4.2f\n", *(brAr + LineNmb), *(thAr + LineNmb));
fprintf(fdOut, "\n %13.6e %13.6e",
        (((*(brAr + LineNmb) * normFactor) - 0.1) * 3),
        (((*(thAr + LineNmb) * normFactor * 10.0) - 0.1) * 2));
```